

# A Gentle Introduction to Static Analysis (I)

Woosuk Lee

CSE 6049 Program Analysis



Hanyang University, Korea

# Comparison between Computing and Other Engineering

	Computing Area	Other Engineering Areas
Object	Software	Machine/building/circuit/chemical process design
Execution subject	Computer runs it	Nature runs it
Our question	Will it work as intended?	Will it work as intended?
Our knowledge	Program analysis	Newtonian mechanics, Maxwell equations, Navier-stokes equations, thermodynamic equations, and other principles

# The Common Goal

	Computing Area	Other Engineering Areas
Object	Software	Machine/building/circuit/chemical process design
Execution subject	Computer runs it	Nature runs it
Our question	Will it work as intended?	Will it work as intended?
Our knowledge	<b>Program analysis</b>	Newtonian mechanics, Maxwell equations, Navier-stokes equations, thermodynamic equations, and other principles

# Our Interest

---

- How to verify specific properties about program executions before execution:
  - absence of run-time errors i.e., no crashes
  - preservation of invariants

## Verification

Make sure that  $\llbracket P \rrbracket \subseteq \mathcal{S}$  where

- **the semantics**  $\llbracket P \rrbracket$  = the set of all behaviors of  $P$
- **the specification**  $\mathcal{S}$  = the set of acceptable behaviors

# Semantics and Semantic Properties

---

## Semantics $\llbracket P \rrbracket$ :

- compositional style (“denotational”)
  - ▶  $\llbracket AB \rrbracket = \dots \llbracket A \rrbracket \dots \llbracket B \rrbracket \dots$
- transitional style (“operational”)
  - ▶  $\llbracket AB \rrbracket = \{s_0 \hookrightarrow s_1 \hookrightarrow \dots, \dots\}$

## Semantic properties $\mathcal{S}$ :

- safety
  - ▶ some behavior observable in *finite* time will never occur.
- liveness
  - ▶ some behavior observable after *infinite* time will never occur.

# Safety Properties

---

- Some behaviors observable in finite time will never occur.
- Examples:
  - No crashing error — e.g., no divide by zero, no uncaught exceptions, etc
  - No invariant violation

# Invariant?

---

- Assertions supposed to be always true
  - e.g., “x has a value larger than 1 at line 5”
- Loop invariant: assertion that holds at the beginning of every loop iteration

```
x = 0;  
while (x < 10) {  
    x = x + 1;  
}
```

Loop invariant 1: “x is an integer”

Loop invariant 2: “ $0 \leq x < 10$ ”

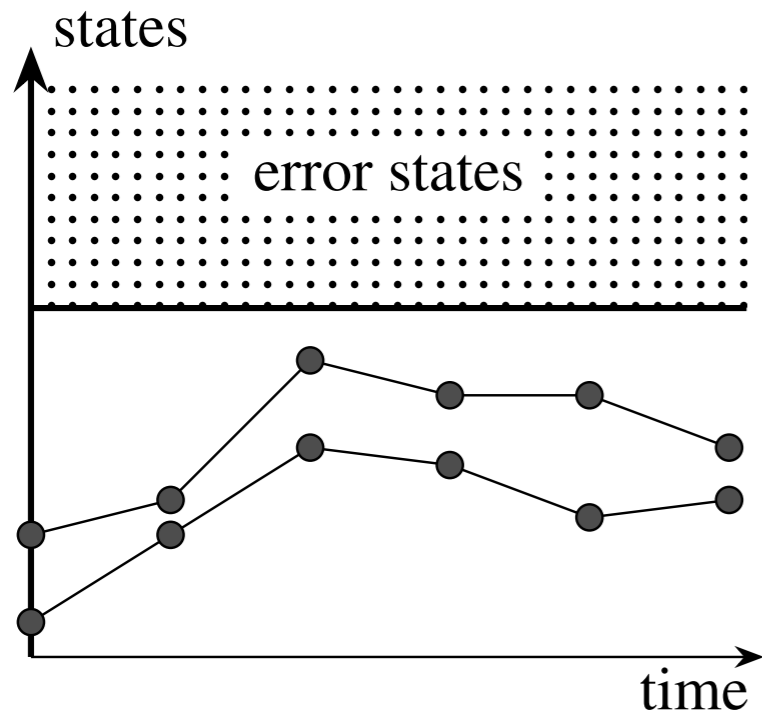
# Example: Division-by-zero

---

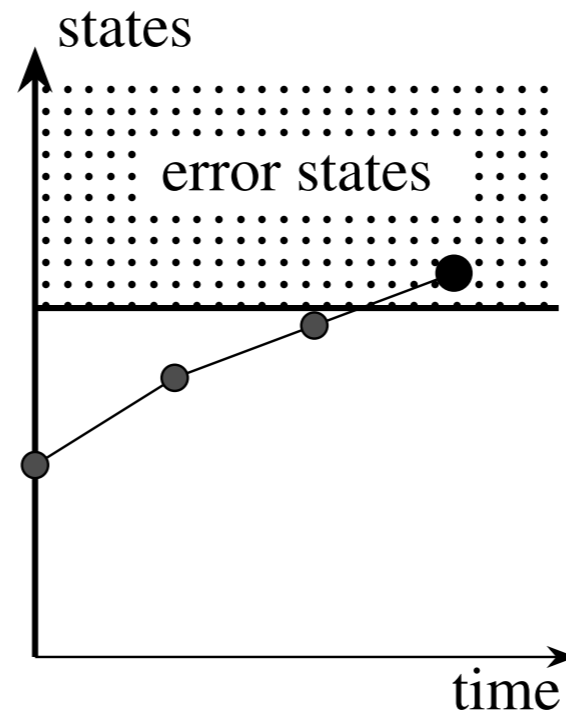
```
1: int main(){
2:     int x = input();           // True
3:     x = 2 * x - 1;             // x is an odd number
4:     while (x > 0) {           // x is a positive odd number
5:         x = x - 2;
6:     }                           // x is an odd number
7:     assert(x != 0);
8:     return 10 / x;
9: }
```



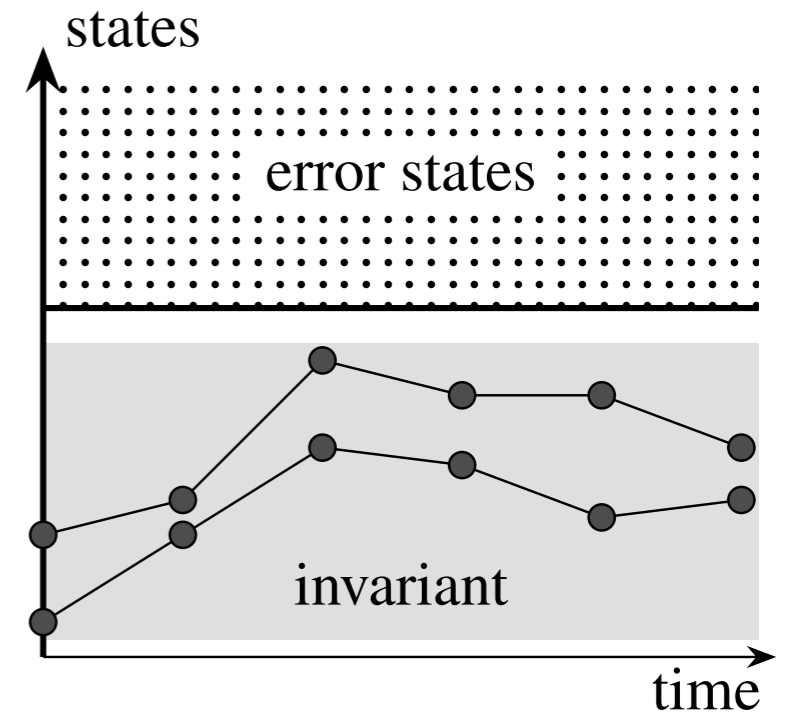
# Proving Safety Properties



(a) Correct executions



(b) An incorrect execution



(c) Proof by invariance

# Liveness Properties

---

- Some behaviors observable after infinite time will never occur
- Examples:
  - No unbounded repetition of a given behavior
  - No non-termination

# Example: proving termination

---

```
x = read_int ();  
while ( x > 0 ) {  
    x = x - 1;  
}
```

- If  $x$  is initially a negative integer  $\Rightarrow$  the program terminates
- If  $x$  is initially a positive integer  $\Rightarrow x$  strictly decreases every iteration  $\Rightarrow$  the program terminates
- In this manner, proves a quantity strictly decreases w.r.t. some well-founded order



A minimum element exists

# A Hard Limit: Undecidability

---

**Theorem (Rice's theorem).** *Any nontrivial semantic properties are undecidable.*

- Semantic property: a property defined wrt the set of executions of a program
- Syntactic property can be decided directly based on the program text.
- Nontrivial property: worth the effort of designing a program analyzer for (trivial: true/false for all programs)

# A Hard Limit: Undecidability

**The Halting Problem:** can we have a function  $H$  that correctly decides if a given program will terminate?

- Answer: No, i.e., we cannot write a function  $H(p)$  that returns true iff program  $p$  terminates.
- *Proof:* proof by contradiction. Suppose we have such a function  $H$ . Consider the following function  
 $f() = \text{if } H(f) \text{ then (while true skip) else skip}$   
Does  $f()$  terminate?  
if  $f()$  terminates, it should not terminate.  
if  $f()$  is non-terminating, it should terminate (contradiction!)

# A Hard Limit: Undecidability

---

- Example: we cannot have an exact analyzer **A** for a property: “This program always prints I and finishes”
- *Proof:* proof by contradiction. Suppose we have such an analyzer.

Given a program **P**, generate **P'** : “**P**; print I”

**A** says “Yes”: **P** halts, **A** says “No”: **P** does not halt

Therefore, we can solve the halting problem

(contradiction!)

# Towards Computability

- More formal version of Rice's theorem:

Common general-purpose languages (e.g., C)

Let  $\mathbb{L}$  be a Turing-complete language, and let  $P$  be a nontrivial semantic property of program of  $\mathbb{L}$ . There exists no *automatic* and *eventually terminating* method such that,

For *every* program  $p$  in  $\mathbb{L}$ , it returns true *if and only if*  $p$  satisfies the semantic property  $P$ .

- We can give up
  - *Automatic*: involving manual efforts
  - *eventually terminating*: possibly nonterminating
  - *Every*: targeting only a restricted class of programs
  - *If and only if*: not always being able to provide an exact answer

# Approximation: Soundness and Completeness

---

- Given a semantic property  $\mathcal{P}$ , and an analysis tool **A**
- If **A** were perfectly accurate,

For all program  $p$ ,  $\mathbf{A}(p) = \text{true} \iff p$  satisfies  $\mathcal{P}$

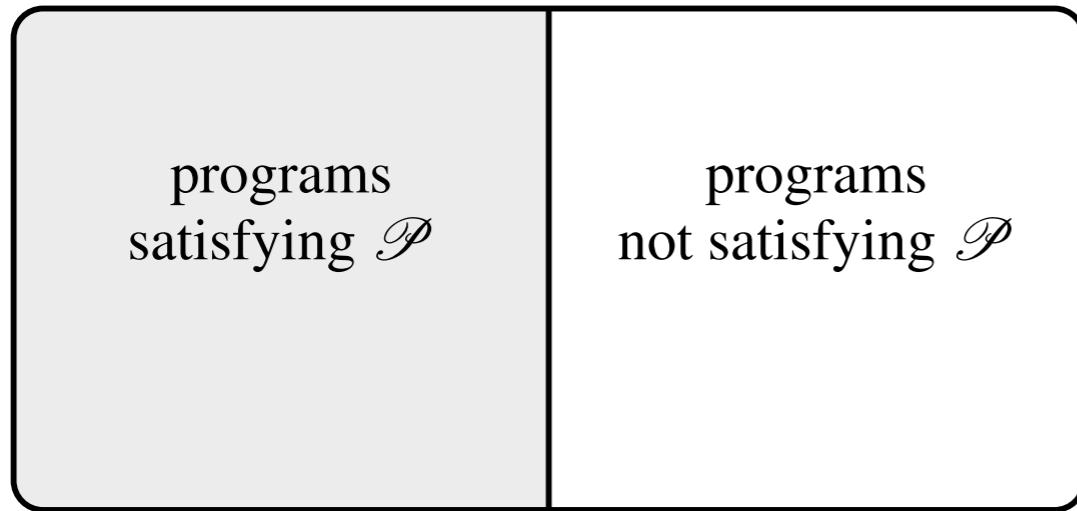
which consists of

For all program  $p$ ,  $\mathbf{A}(p) = \text{true} \implies p$  satisfies  $\mathcal{P}$       **(soundness)**

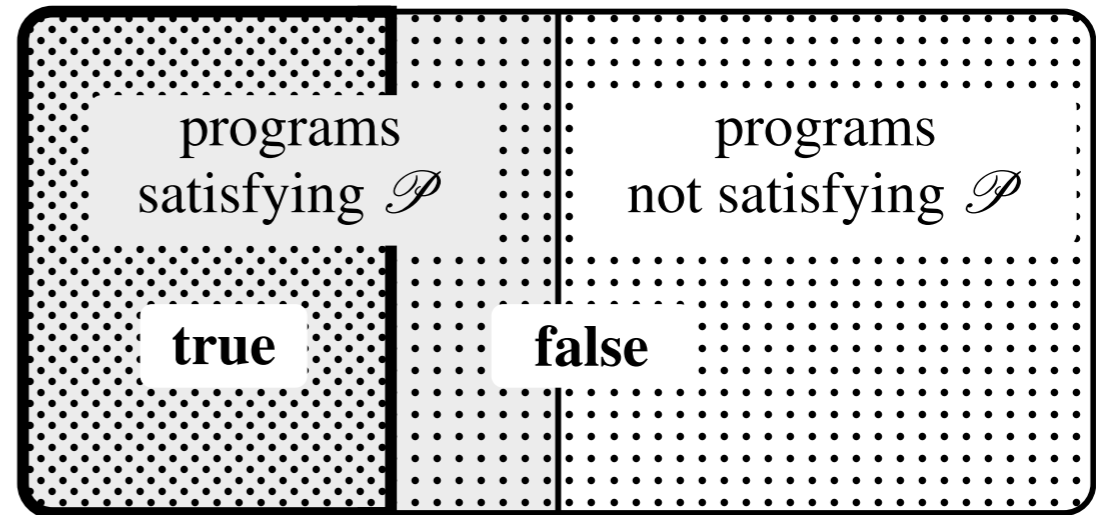
For all program  $p$ ,  $\mathbf{A}(p) = \text{true} \Leftarrow p$  satisfies  $\mathcal{P}$       **(completeness)**



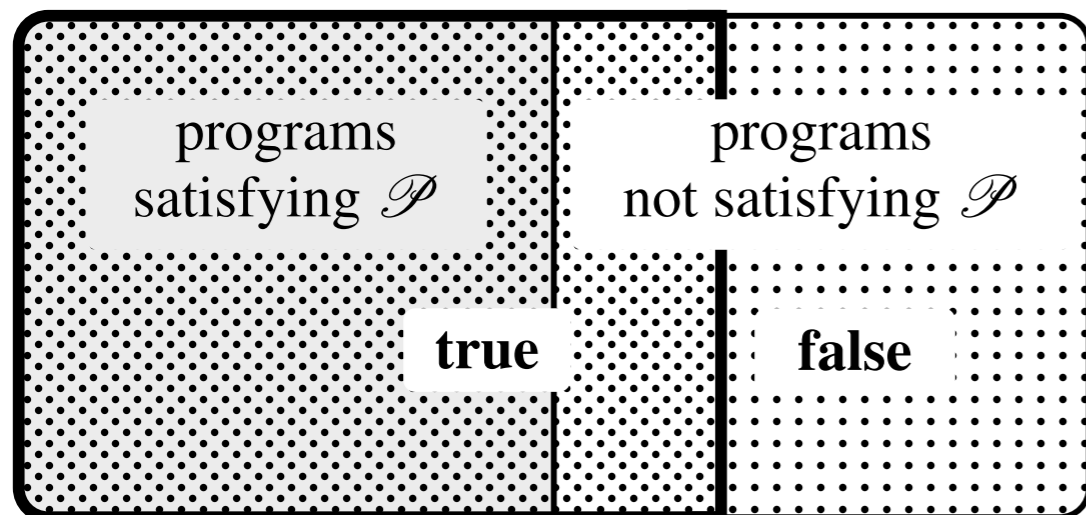
# Approximation: Soundness and Completeness



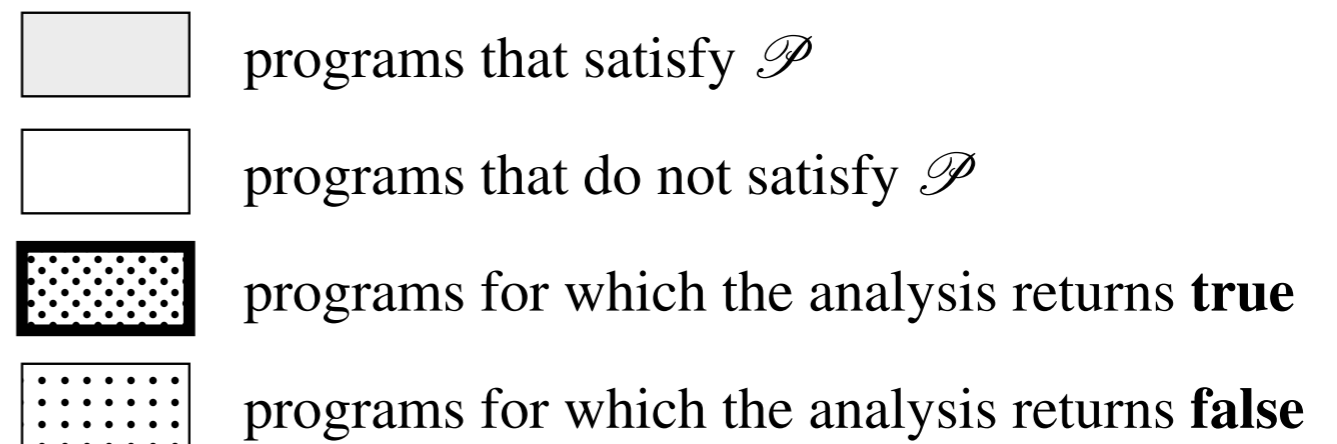
(a) Programs



(b) Sound, incomplete analysis



(c) Unsound, complete analysis



(d) Legend

# Spectrum of Program Analysis Techniques

---

- Testing
- Machine-assisted proving
- Finite-state model checking
- Conservative static analysis
- Bug-finding

# Testing

---

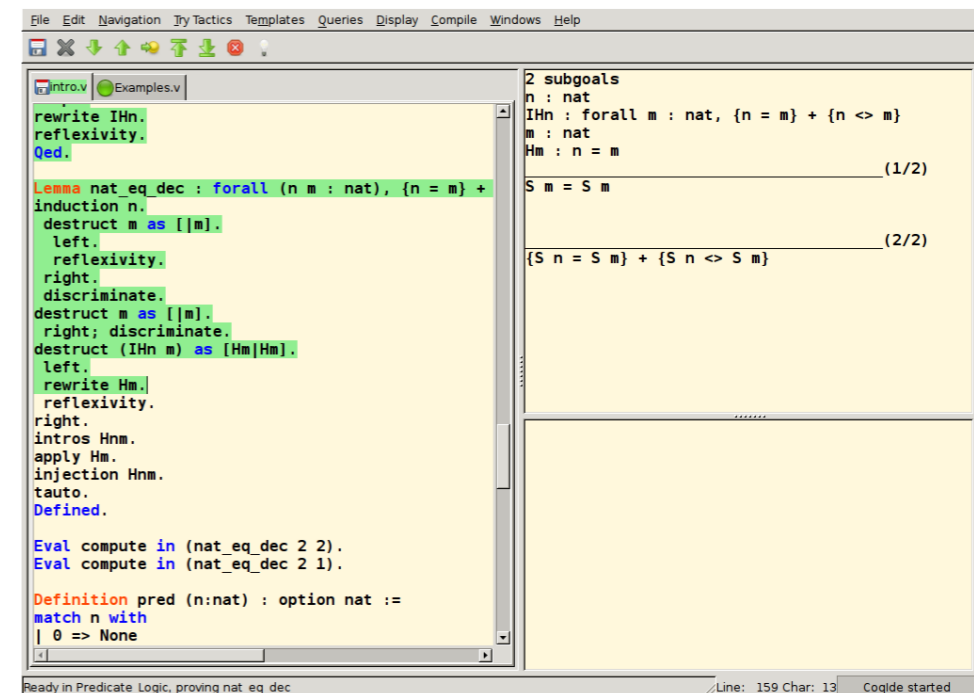
- Consider finitely many, finite executions
- For each of them, check whether it violates the spec. If the finite executions find no bug, then accept.
- **Unsound**: can accept programs violating the spec
- **Complete**: does not reject programs that satisfy the spec

# Machine-assisted Proving

- Use a specific language to formalize verification goals, manually supply proof arguments, and let the proofs be automatically verified

- Tools: Coq, Isabelle/HOL, PVS, ...

- Applications:  
CompCert (certified compiler),  
seL4 (secure micro-kernel), ...



The screenshot shows the Coq proof assistant interface. The main window displays a proof script for a lemma named `nat_eq_dec`. The script includes tactics such as `rewrite IHn`, `reflexivity`, `Qed`, `Lemma`, `induction`, `destruct`, `left`, `right`, `discriminate`, `intros`, `apply`, `injection`, `tauto`, and `Defined`. It also shows evaluation commands and a definition for a predicate `pred`. The right-hand pane shows the current goals, including subgoals for `n : nat`, `IHn : forall m : nat, {n = m} + {n <> m}`, and `Hm : n = m`. The status bar at the bottom indicates the current position in the file and that CoqIDE is started.

- **Not automatic**: key proof arguments provided by users
- **Sound**: if the formalization is correct
- **Quasi-complete**: (only limited by the expressiveness of the logics)

# Finite-State Model Checking

---

- Focus on finite state models of programs
- Perform exhaustive exploration of program states
- **Automatic, sound & complete** (only wrt the finite models)
- **May not terminate:** SW has infinitely many states: the models need approximation or non-termination.

# Conservative Static Analysis

---

- Perform automatic verification, yet which may fail
- Compute a conservative approximation of the program semantics
- **Automatic & Sound:** accepted programs are safe
- **Incomplete:** may reject safe programs (false alarms)
- Analysis algorithms reason over program semantics
- Examples: Astree, Sparrow, Facebook Infer, ...

# Bug Finding

---

- Commercial tools: Coverity, CodeSonar, ...
- Automatic and fast
- **Unsound**: may accept an incorrect program
- **Incomplete**: may reject a correct program
- Used to increase SW quality without any guarantee

# Comparison

---

	automatic	sound	complete
testing	yes	no	yes
machine-assisted proving	no	yes	yes/no
finite-state model checking	yes	yes/no	yes/no
conservative static analysis	yes	yes	no
bug-finding	yes	no	no



# Focus of This Course: Conservative Static Analysis

---

A general technique, for any programming language  $\mathbb{L}$  and safety property  $\mathcal{S}$ , that

- **checks**, for input program  $P$  in  $\mathbb{L}$ , if  $\llbracket P \rrbracket \subseteq \mathcal{S}$ ,
- **automatic** (software)
- **finite** (terminating)
- **sound** (guarantee)
- **malleable** for arbitrary precision